Chapter 8 Software testing:

8.1 Development testing

8.2 Test-driven development

8.3 Release testing

8.4 User testing

8.1 Development testing:

Overview

Development testing encompasses all testing activities conducted by the development team. The tester is often the programmer who wrote the software. Development testing serves mainly as a defect testing process aimed at bug discovery and is closely related to debugging.

Testing Approaches

- For standard projects, the developer often acts as the tester.
- In some methodologies, each programmer has an associated tester who assists in the testing process.
- For critical systems, a dedicated testing group within the development team may handle testing.

Debugging

- Debugging is the action of correcting errors found during testing.
- Debuggers use specialized interactive tools to provide additional program execution information.
- Debugging is typically interleaved with development testing.

Stages of Development Testing

1. **Unit Testing**
    - Focuses on individual program units or object classes.
    - The main objective is to test the functionality of objects or methods.
2. **Component Testing**
    - Involves testing integrated units to form composite components.

- Focuses on testing the interfaces that enable access to component functions.

3. **System Testing**
    - Involves testing the system as a whole after integrating some or all components.
    - Focuses on testing interactions between components.

Conclusion

Development testing is a critical part of the software development lifecycle. It is primarily concerned with defect discovery and is closely related to debugging. The approach to development testing may vary depending on the criticality of the system.

8.1.1 Unit testing

Overview

Unit testing focuses on testing individual program components like methods or object classes. It aims for thorough coverage of all features of the object under test.

Key Aspects of Unit Testing

1. **Test Design**
    - Your tests should include calls to the routines with varying input parameters.
    - For object classes, strive to test all operations, attributes, and states associated with the object.
2. **Testing Sequences**
    - Some methods may require preconditions to be met or other methods to be executed first.
    - For example, to test a `shutdown` method, a `restart` method might need to be executed first.
3. **Inheritance Complications**
    - Inheritance can complicate testing, as operations in parent classes may make assumptions that are not valid in subclasses.
    - Inherited operations need to be tested in every subclass where they are used.
4. **State Testing**
    - Utilize state models to identify sequences of state transitions for testing.
    - Example sequences include "Shutdown → Running → Shutdown" and "Configuring → Running → Testing → Transmitting → Running".
5. **Automated Unit Testing**
    - Make use of test automation frameworks like JUnit.
    - Automated tests typically consist of three parts:
        a. Setup: Initialize the system with the test case.
        b. Call: Execute the object or method to be tested.
        c. Assertion: Compare the result with the expected outcome.
6. **Mock Objects**
    - Used when the object under test has dependencies that are not yet implemented or would slow down testing.
    - Mock objects simulate the functionality of real objects but are faster to set up and execute.

Conclusion

Unit testing is a critical and detailed activity that aims to validate the functionality of individual components. It often employs automated frameworks and may use mock objects to handle dependencies.

8.1.2 Choosing unit test cases

Objectives of Effective Test Cases

1. Confirm that the component functions as expected.
2. Uncover any defects in the component.

Types of Test Cases

1. **Normal Operation Cases**: Test if the component functions as expected under normal conditions.
2. **Abnormal Operation Cases**: Use abnormal inputs to check that they are properly processed and do not crash the component.

Strategies for Test Case Selection

1. **Partition Testing**
    - Identify groups of inputs with common characteristics.
    - Design test cases so that inputs lie within these partitions.
    - Rule of thumb: Choose test cases on the boundaries and close to the midpoint of each partition.
2. **Guideline-based Testing**
    - Use testing guidelines reflecting previous experience of common errors.

Input and Output Partitions

- Input and output data can be categorized into sets with common characteristics, known as equivalence partitions.
- Partition testing can be applied to both system and component levels.

Black-box and White-box Testing

- **Black-box Testing**: Based solely on the program specification.
- **White-box Testing**: Involves looking at the code to find other possible tests, such as exception handling.

Testing Guidelines for Sequences, Arrays, or Lists

1. Test with sequences that have only a single value.
2. Use sequences of different sizes in different tests.
3. Access the first, middle, and last elements of the sequence.

General Testing Guidelines

- Choose inputs that force the system to generate all error messages.
- Design inputs that cause input buffers to overflow.
- Repeat the same input or series of inputs multiple times.
- Force invalid outputs to be generated.

- Force computation results to be too large or too small.

Conclusion

Choosing effective unit test cases is crucial for the verification of component functionality and the discovery of defects. The use of partition testing and guideline-based testing can aid in this selection process.

8.1.3 Component testing

Focus of Component Testing

- Testing targets the component interfaces, assuming unit tests on individual objects are complete.
- Interface errors in composite components may arise from interactions between objects.

Types of Interfaces and Errors

1. **Parameter Interfaces**
   - Data or function references are passed between components.
   - Common errors: wrong type, wrong order, or wrong number of parameters.
2. **Shared Memory Interfaces**
   - Memory is shared between components, often in embedded systems.
   - Common error: timing issues leading to outdated information.
3. **Procedural Interfaces**
   - One component provides a set of procedures for others to call.
   - Common error: differing failure assumptions or misunderstandings about behavior.
4. **Message Passing Interfaces**
   - Components communicate through messages, common in client–server systems.
   - Common error: timing problems due to different speeds of producer and consumer.

Classes of Interface Errors

1. **Interface Misuse**: Incorrect use of a component's interface by the calling component.
2. **Interface Misunderstanding**: The calling component misunderstands the called component's behavior.
3. **Timing Errors**: Occur in real-time systems, particularly with shared memory or message-passing interfaces.

General Guidelines for Interface Testing

1. Test with extreme values of parameter ranges to reveal inconsistencies.
2. Always test with null pointer parameters where pointers are passed.
3. Deliberately cause component failure through procedural interfaces to test assumptions.
4. Use stress testing in message-passing systems to reveal timing problems.
5. Vary the activation order of components interacting through shared memory.

Alternatives to Testing

- Inspections and reviews can sometimes be more effective for detecting interface errors.

Summary

Component testing is crucial for verifying the correct interaction between different parts of a system, focusing primarily on the interfaces. Different types of interfaces are susceptible to different kinds of errors, often arising from misunderstandings or misuse. Carefully designed testing strategies, sometimes supplemented by inspections, can help in identifying and rectifying these errors.

8.1.4 System testing

Objectives and Scope

- Integrates and tests the complete system to check component compatibility and correct interaction.
- Distinguishes itself from component testing by including reusable and off-the-shelf components and by being a collective process.

Types of Behavior

1. **Planned Emergent Behavior**: Tested to ensure the system performs as designed when components are integrated.
2. **Unplanned Emergent Behavior**: Tests are developed to make sure the system only does what it is intended to do.

Focus Areas

- Emphasis on testing interactions between components and objects.
- Use case-based testing is effective for this purpose, as it naturally involves multiple components.

Use Case and Sequence Diagrams

- Useful for designing specific test cases, as they show required inputs and expected outputs.

Challenges and Strategies

- Difficult to determine the extent of system testing needed.
- Exhaustive testing is impossible; testing is based on a subset of possible test cases.
- Incremental integration and testing is recommended.

Testing Policies

1. Test all system functions accessed through menus.
2. Test combinations of functions accessed through the same menu.
3. Test all functions with both correct and incorrect user input.

Automation Limitations

- More difficult than automated unit or component testing due to unpredictable outputs.

Summary

System testing is a comprehensive process that aims to validate the integrated system as a whole, focusing on both planned and unplanned emergent behavior. It leverages use case-based testing and sequence diagrams to create effective test cases. The process faces challenges in determining the necessary extent and type of tests, often relying on incremental approaches and specific testing policies.

8.2 Test-driven development

Fundamental Process

1. **Identify Functionality**: Start by identifying a small, implementable increment of functionality.
2. **Write Test**: Create an automated test for the functionality.
3. **Run Test**: Initially, the test will fail as the functionality is not yet implemented.
4. **Implement Functionality**: Write code for the functionality and possibly refactor existing code.
5. **Pass All Tests**: Ensure the new and existing tests all pass before moving on to the next functionality increment.

Essential Tools

- An automated testing environment, such as JUnit, is crucial for TDD due to the frequent running of tests.

Advantages

1. **Code Coverage**: Ensures all code segments are tested, identifying defects early.
2. **Regression Testing**: Builds a test suite incrementally, allowing for easy regression tests.
3. **Simplified Debugging**: When a test fails, the issue is likely in the newly written code, minimizing debugging effort.
4. **System Documentation**: Tests act as documentation, explaining what the code is supposed to do.

Limitations

- **Incomplete Knowledge**: If you don't know enough to write tests, you won't develop the required code.
- **Large Component Reuse**: Not suitable for scenarios where large code components or legacy systems are being reused.
- **Multithreaded Systems**: May produce inconsistent results due to interleaved threads.

Regression Testing Costs

- TDD significantly reduces the costs associated with regression testing by utilizing automated tests.

Applicability

- Useful in both agile and plan-based development processes.
- Does not eliminate the need for system testing.

Overall Effectiveness

- Widely accepted and considered productive by most programmers.

- Claims about improved code structure and quality are still inconclusive.

Summary

Test-Driven Development (TDD) is an iterative process that integrates coding with automated testing. It has benefits like early defect detection and simplified debugging but has limitations such as being less effective for legacy or multithreaded systems. It is a mainstream practice but still requires system testing for full validation.

8.3 Release testing

Definition

- Release testing is the final testing phase for a system intended for external use, be it customers, other development teams, or product management.

Key Distinctions from System Testing

1. **Responsibility**: The development team should not conduct release testing.
2. **Purpose**: While system testing aims to discover bugs, release testing focuses on validating that the system meets its requirements and is fit for use by the customers.

Primary Goal

- The main objective is to convince the supplier that the system is good enough for release. It must deliver on its specified functionality, performance, and dependability, and not fail during normal use.

Testing Approach

- Typically, employs black-box testing, where tests are based on the system specification.
- Also known as functional testing, as the focus is solely on the system's functionality and not its implementation.

Summary

Release testing is the validation phase that ensures a system is ready for external use. It is distinct from system testing, primarily in its responsibility and purpose. The approach commonly used is black-box or functional testing.

8.3.1 Requirements-based testing

Definition

- Focuses on ensuring each requirement is testable and then designing a test case for it.
- The aim is to validate that the system has properly implemented its requirements.

Testing Approach

- Derive a set of tests for each requirement.

- Typically, involves multiple tests for each requirement to ensure full coverage.

Example Scenario

For a Mentcare system concerned with drug allergies, several related tests may need to be developed:

1. **No Allergies**: Create a patient record with no known allergies and prescribe medication. Verify that no warning message is issued.
2. **Known Allergy**: Create a patient record with a known allergy and prescribe that medication. Verify that a warning is issued.
3. **Multiple Allergies**: Create a patient record with allergies to two or more drugs. Prescribe these drugs separately and verify that the correct warnings are issued.
4. **Multiple Warnings**: Prescribe two drugs that the patient is allergic to and verify that two warnings are issued.
5. **Overrule Warning**: Prescribe a drug that triggers a warning, overrule it, and verify that the system prompts for a reason for the override.

Traceability

- Maintain records linking tests to specific requirements to ensure each requirement is adequately tested.

Summary

Requirements-based testing is a systematic approach to validate the implementation of each system requirement. It often involves multiple test cases per requirement for comprehensive coverage and maintains traceability records.

8.3.2 Scenario testing

Definition

- Scenario testing is an approach to release testing using typical scenarios or stories that describe one way the system might be used.
- Targets real-world use cases to develop test cases for the system.

Characteristics of a Good Scenario

- Should be a credible, fairly complex narrative story.
- Should be relatable to stakeholders.

Example Scenario: Mentcare System

The scenario involves a nurse named George who uses the Mentcare system during home visits to patients. The scenario tests multiple features:

1. **Authentication**: Logging into the system.
2. **Data Transfer**: Downloading and uploading patient records to a laptop.
3. **Scheduling**: Managing home visit schedules.

4. **Encryption/Decryption**: Secure handling of patient records on a mobile device.
5. **Record Modification**: Retrieval and modification of patient records.
6. **Drug Database**: Interfacing with a database that maintains drug side-effect information.
7. **Call Prompting**: Generating a call list for follow-up actions.

Testing Approach

- Role-play the scenario and go through it step-by-step.
- Deliberately introduce errors to test the system's robustness.
- Note any issues, including performance problems.

Benefits

- Tests multiple requirements within a single scenario.
- Helps in checking that combinations of requirements do not cause issues.
- Useful for identifying usability and performance issues that might change real-world use.

Summary

Scenario testing involves using realistic, complex stories that stakeholders can relate to for release testing. It enables the testing of multiple features and their combinations, often revealing issues not easily caught by isolated requirement-based tests.

8.3.3 Performance testing

Objectives

- Test emergent properties like performance and reliability once the system is completely integrated.
- Aim to ensure that the system can handle its intended load.

Two-Fold Focus

1. **Demonstrate Compliance**: Show that the system meets its performance requirements.
2. **Defect Identification**: Discover problems and defects, particularly those that manifest under heavy load.

Operational Profile

- A set of tests reflecting the actual mix of work the system will handle.
- For example, if 90% of transactions are of type A, the operational profile should predominantly test type A transactions to get an accurate measure of performance.

Stress Testing

- Purposefully overload the system to test its limits.
- Gradually increase the load beyond the system's maximum design limit to test how it behaves under failure conditions.

Goals of Stress Testing

1. **Test Failure Behavior**: Ensure that system failure under extreme conditions does not result in data corruption or loss of services. Aim for a "fail-soft" rather than a collapse.
2. **Reveal Hidden Defects**: Identify defects that may only show up when the system is fully loaded.

Applicability

- Especially relevant for distributed systems where network load can significantly affect performance.
- Helps identify the point at which system degradation begins, allowing for transaction rejection beyond this point.

Summary

Performance testing is crucial for ensuring that a system can handle its intended load and for identifying potential defects. It often involves the use of an operational profile for realistic testing and employs stress testing to examine the system's behavior under extreme conditions. This is particularly important for distributed systems where performance degradation can be a major issue.

8.4 User testing

Overview

- User testing involves customers or users providing input and advice on system testing.
- Essential for capturing influences from the user’s working environment that can affect system reliability, performance, and usability.

Types of User Testing

1. **Alpha Testing**: A selected group of users works closely with developers to test early releases.
    - Allows for early identification of problems and issues not apparent to the development team.
    - Commonly used in both product development and agile methodologies.
2. **Beta Testing**: A larger group of users tests an early release of the software.
    - Used to discover interaction problems between the software and its operational environment.
    - Also serves as a form of marketing.
3. **Acceptance Testing**: Customers test the system to decide if it's ready for deployment.
    - Integral part of custom systems development.
    - Outcome may lead to conditional acceptance of the system.

Stages in Acceptance Testing

1. **Define Acceptance Criteria**: Should be part of the system contract and approved by both parties.
2. **Plan Acceptance Testing**: Involves resource allocation, time, budget, and risk mitigation strategies.
3. **Derive Acceptance Tests**: Aim to cover both functional and nonfunctional characteristics of the system.
4. **Run Acceptance Tests**: Ideally conducted in the actual environment where the system will be used.
5. **Negotiate Test Results**: Developer and customer negotiate whether the system is good enough to be used.
6. **Reject/Accept System**: A final decision is made based on test outcomes and negotiations.

Agile Methods

- In agile methods like Extreme Programming, the end-user is part of the development team.
- These users should ideally be "typical" users, but finding such users can be challenging.

Challenges and Limitations

- It's difficult for developers to replicate the user’s actual working environment.
- Requirement for automated testing in agile methods limits the flexibility of interactive systems.

Summary

- Testing can only show the presence of errors in a program. It cannot show that there are no remaining faults.
- Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers. In the user testing process, customers or system users provide test data and check that tests are successful.
- Development testing includes unit testing in which you test individual objects and methods; component testing in which you test related groups of objects; and system testing in which you test partial or complete systems.
- When testing software, you should try to “break” the software by using experience and guidelines to choose types of test cases that have been effective in discovering defects in other systems.
- Wherever possible, you should write automated tests. The tests are embedded in a program that can be run every time a change is made to a system.
- Test-first development is an approach to development whereby tests are written before the code to be tested. Small code changes are made, and the code is refactored until all tests execute successfully.
- Scenario testing is useful because it replicates the practical use of the system. It involves inventing a typical usage scenario and using this to derive test cases.
- Acceptance testing is a user testing process in which the aim is to decide if the software is good enough to be deployed and used in its planned operational environment.